BLACK CODHER

CODING PROGRAMME

CODING BLACK FEMALES

Black Codher Bootcamp

NÍ YÓ

BLACK CODHER

CODING PROGRAMME

UNIT 4 - Session 3
React

Last Session:

In the last session we covered:

1. Understand what React components are and how React uses a virtualDOM instead of the real DOM (JQuery, Vanilla Javascript)

2. Understand what package managers are and the different types (NPM, NPX, YARN)

3. Understand Node.js, download and install it

4. Create your first React project using the create-react-app command

5. Understand what files make up a React web application

6. Understand and run a unit test using the Jest test runner

Goals for Unit 4 - Session 3

1. Edit the Create React App
2. View the React elements in a browser window (using Developer Tools)
3. Class components and Functional Components
4. JSX Components Explained
5. Understand Component Lifecycle
6. Understand Props (Properties) and States in React

4

Editing the Create React App

5

Editing the App: index.html

- In the **index.html** you can see the ** ** tag with an id of root. This is an index.html file so it contains real HTML.

- Notice that the ** ** tag is the only visible tag in the body (line 31).

```
22
23        Unlike "/favicon.ico" or "favicon.ico", "%PUBLIC_URL%/favicon.ico" will
24        work correctly both with client-side routing and a non-root public URL.
25        Learn how to configure a non-root public URL by running `npm run build`.
26      -->
27      <title>CBF First React Application</title>
28    </head>
29    <body>
30      <noscript>You need to enable JavaScript to run this app.</noscript>
31      <div id="root"></div>
32      <!--
33        This HTML file is a template.
34        If you open it directly in the browser, you will see an empty page.
```


6

Editing the App: index.html

- The ** ** tag on line 31 is the main anchor tag where your components will be rendered in a React App.

- Open **index.js** to understand how the app code is rendered.

```
22
23        Unlike "/favicon.ico" or "favicon.ico", "%PUBLIC_URL%/favicon.ico" will
24        work correctly both with client-side routing and a non-root public URL.
25        Learn how to configure a non-root public URL by running `npm run build`.
26      -->
27      <title>CBF First React Application</title>
28    </head>
29    <body>
30      <noscript>You need to enable JavaScript to run this app.</noscript>
31      <div id="root"></div>
32      <!--
33        This HTML file is a template.
34        If you open it directly in the browser, you will see an empty page.
```


7

Editing the App: index.js

- The function call in this JavaScript file is the ReactDOM which calls one method **render()** and passes a value of **<App/>** (Line 7-12)

- **<App/>** is an example of a **JSX** component.

- The App component is defined in the file **App.js**.

- The **import** statement is used to import read only **live bindings*** which are exported by another module

App.js | index.html | index.js

src > index.js

```
1  import React from 'react';
2  import ReactDOM from 'react-dom';
3  import './index.css';
4  import App from './App';
5  import * as serviceWorker from './serviceWorker';
6
7  ReactDOM.render(
8    <React.StrictMode>
9      <App />
10   </React.StrictMode>,
11   document.getElementById('root')
12 );
```


8

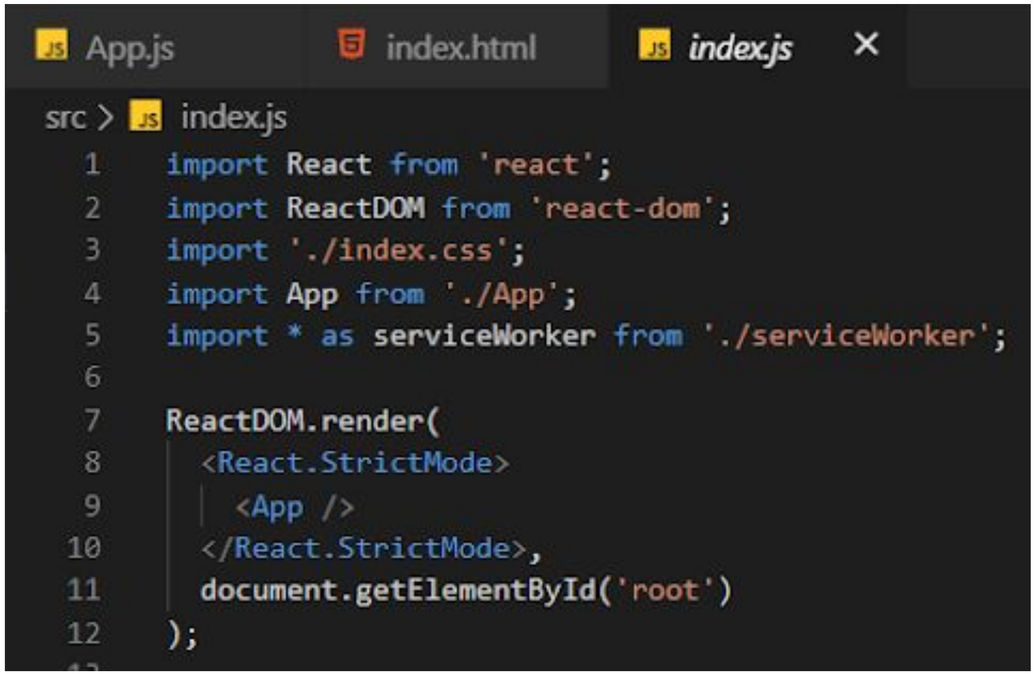
Editing the App: index.js

- Line 4 shows the App component being imported into the current file (the extension .js is assumed by the React compiler).

- Note the import syntax:

```
import <object name> from '<object path>';
```

Note: **<React.StrictMode>** (line 8 and 10) is a special tag that does not render anything to screen. It's used as a way of validating the components underneath it.



9

Editing the App: index.js

- **Import**ing ‘react’ and the ‘react-dom’ is essential for creating React Apps!

- **react** and **react-dom** have been installed in the local **node_modules** folder so do not require a relative path.

- Next, let's take a look at the specific App component in **./App.js**.

App.js | index.html | index.js

src > index.js

```
1  import React from 'react';
2  import ReactDOM from 'react-dom';
3  import './index.css';
4  import App from './App';
5  import * as serviceWorker from './serviceWorker';
6
7  ReactDOM.render(
8    <React.StrictMode>
9      <App />
10   </React.StrictMode>,
11   document.getElementById('root')
12 );
```

Editing the App: App.js

- Line 7 to 22 shows some markup. It's important to note that although the markup looks like HTML it is not.

- This is an example of a **Functional Component** returning **JSX**.

- **JSX** is **J**ava**S**cript **XML** (Extensible Markup Language).

src > App.js > App

```
1  import React from 'react';
2  import logo from './logo.svg';
3  import './App.css';
4
5  function App() {
6    return (
7      <div className="App">
8        <header className="App-header">
9          <img src={logo} className="App-logo" alt="logo" />
10         <p>
11           Edit <code>src/App.js</code> and save to reload.
12         </p>
13         <a
14           className="App-link"
15           href="https://reactjs.org"
16           target="_blank"
17           rel="noopener noreferrer"
18         >
19           Learn React
20         </a>
21       </header>
22     </div>
23   );
24 }
25
26 export default App;
27
```


11

Editing the App: App.js

- Since **JSX** is closer to **JavaScript** than to **HTML**, the React DOM uses camelCase property naming conventions instead of HTML attribute names.

- E.g. *class** becomes class*Name* in JSX, and *tabindex* becomes tab*Index* when called from JSX.

src > App.js > App

```
import React from 'react';
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App;
```


12

Editing the App: App.js

- The App component is composed of other **JSX** elements, e.g.
  - div (line 7-22)
  - header (line 8-21)
  - p - paragraph, a- anchor tags

These are all **JSX** elements **not HTML**

- Unlike browser **DOM** elements, React elements are plain objects, and are cheap to create.

- React DOM takes care of updating the DOM to match the React elements.

src > App.js > App

```
1  import React from 'react';
2  import logo from './logo.svg';
3  import './App.css';
4
5  function App() {
6    return (
7      <div className="App">
8        <header className="App-header">
9          <img src={logo} className="App-logo" alt="logo" />
10         <p>
11           Edit <code>src/App.js</code> and save to reload.
12         </p>
13         <a
14           className="App-link"
15           href="https://reactjs.org"
16           target="_blank"
17           rel="noopener noreferrer"
18         >
19           Learn React
20         </a>
21       </header>
22     </div>
23   );
24 }
25
26 export default App;
27
```


13

Updating the Create-React-App

- If we change **index.js**: update line 5 in the application will allow the component to be render and displayed in the root element.

- The root element is in the **index.html** file.

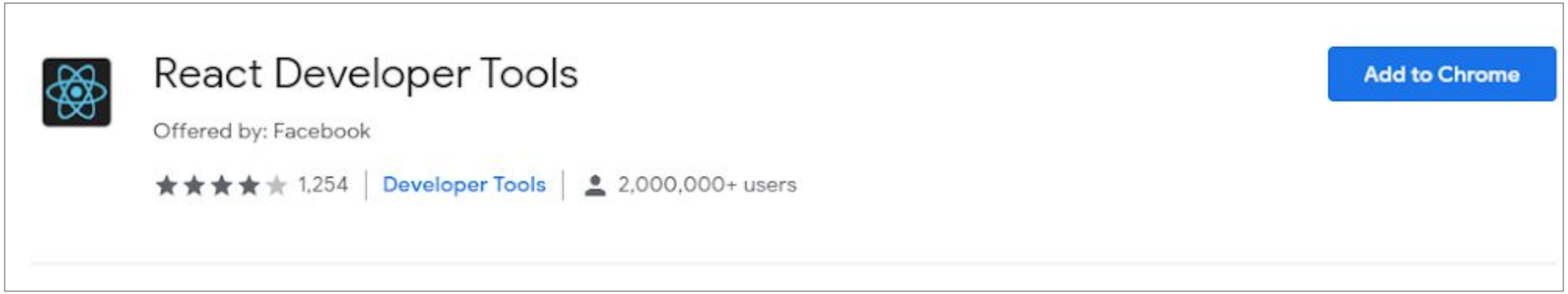
- In the browser http://localhost:3000 you can now see the **<h1>** markup.

```
1  import React from 'react';
2  import ReactDOM from 'react-dom';
3
4
5  const element = <h1>My Reading List</h1>;
6  console.log(element);
7  ReactDOM.render(element,document.getElementById('root'));
```

```
22
23      Unlike "/favicon.ico" or "favicon.ico", "%PUBLIC_URL%/favicon.ico" will
24      work correctly both with client-side routing and a non-root public URL.
25      Learn how to configure a non-root public URL by running `npm run build`.
26    -->
27    <title>CBF First React Application</title>
28  </head>
29  <body>
30    <noscript>You need to enable JavaScript to run this app.</noscript>
31    <div id="root"></div>
32    <!--
33      This HTML file is a template.
34      If you open it directly in the browser, you will see an empty page.
```

Installing React Developer Tools

- Install the React Developer Tools extension.
- The extensions adds React debugging tools to the Chrome Developer Tools.
- Visit: https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=en



15

Inspecting the Elements

16

Viewing the React Output

- When running the project (`> npm start`), if you open in Chrome, it is possible to see the object output of the **JSX** element.

- In the snippet below you can see the type property is set to "h1". The props (content of the h1) is set to 'My Reading List'.

```
▼ {$$typeof: Symbol(react.element), type: "h1", key: null, ref: null, props: {…}, …}
    $$typeof: Symbol(react.element)
    key: null
  ▶ props: {children: "My Reading List"}
    ref: null
    type: "h1"
    _owner: null
  ▶ _store: {validated: false}
    _self: null
```


17

Viewing the React Output

- The React element is part of the Virtual DOM.
- When the state of the element changes React will recognize the change and find the corresponding element in the real DOM and update accordingly.

```
▼ {$$typeof: Symbol(react.element), type: "h1", key: null, ref: null, props: {…}, …}
    $$typeof: Symbol(react.element)
    key: null
  ▶ props: {children: "My Reading List"}
    ref: null
    type: "h1"
    _owner: null
  ▶ _store: {validated: false}
    _self: null
```


18

Checkpoint!

How are you feeling?

RED - I have no idea what you're talking about.

YELLOW - I have some questions but feel like I understand some things.

GREEN - I feel comfortable with everything you've said.

Exercise 1

20

Exercise 1: Updating React App

1. Open your create-react-app test project and delete the contents of the **src** file.

2. Add a new **index.js** file under **src** folder and add the following lines of code:

```
import React from 'react';

import ReactDOM from 'react-dom';

const element = <h1>Welcome to the Book Library</h1>;

console.log(element);
```

Line 1 and 2 import the React and ReactDOM into the project. Line 4 declares a simple JSX element and line 5 logs the element to the console window.

21

Exercise 1: Updating React App

BLACK CODHER
CODING PROGRAMME

3. Run the the app by calling **npm start**. This will open the project in a new browser window. Alternatively open **http://localhost:3000** to see changes.

4. The browser window should be empty as you are not rendering anything to screen. Open the developer tools in the browser (**Ctrl + Shift + i** in Chrome) or open the developer tools in whatever browser you are in. Go to the console. You should be able to see the output of the element

5. Add the following lines of code to render the element to screen:

```
ReactDOM.render(element, document.getElementById('root'));
```

6. Call **npm start** from the command line to see the changes.

22

Exercise 1: Create React App

1. Open the React App you created in the previous session or run the command: > `npx create-react-app myproject` to create a new project

2. Open the **index.html** file under the public folder and change the title tag to "React Exercise 1". Refresh your browser window to view changes

3. Open the App.css file under the src directory and update the application background colour

4. Open **App.js** and change the displayed text by removing the **<p>** and **<a>** elements and adding your own **<h1>** element

23

JSX

24

JSX

- JSX stand for **J**avaScript e**X**tensible markup language.

- JSX is a syntax extension to JavaScript. It is optional and not required to use in React!

Why use JSX?

- JSX produces React "elements".

- Each React element is encapsulated meaning it can operate independently as all the needed data (functions, properties, etc.) are together in a single unit.

25

Why Use JSX

BLACK CODHER
CODING PROGRAMME

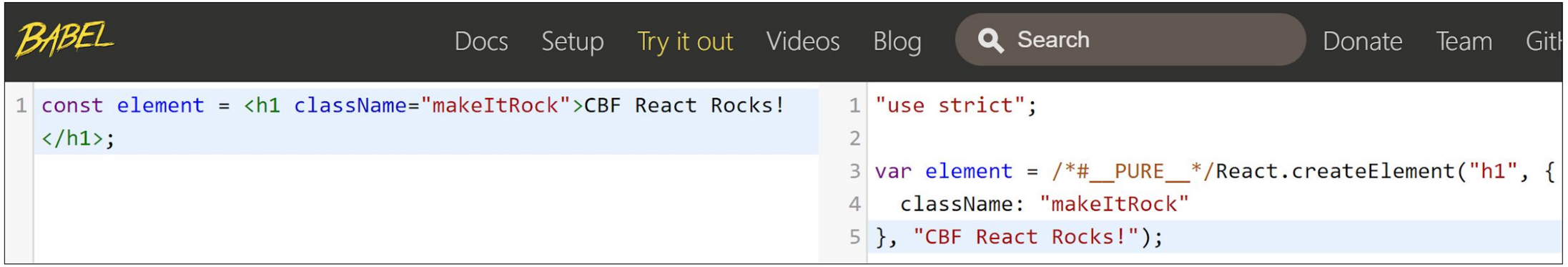
- Each React element is a JavaScript object that you can store in a variable or pass around in your program. This can make your code more **reusable**
- React does not require **JSX** but it can be helpful as a visual aid when working with UI (User Interface) inside JavaScript code
- Using **JSX** also allows for more useful error and warning messages
- It’s important to note that JSX "**transpiles**" (translate and compiles) to plain JavaScript **React.createElement()** calls
- Next we’ll take a quick look at how **JSX** is converted to plain old React elements

26

Babel and Plain Old React Code

- Babeljs.io is a library that is used in React to translate **JSX** to plain old React code.

- Visit https://babeljs.io/repl to test the compiler.



- It can be helpful to see how babel translates React class components vs react functional components

- The example above show how a plain old JSX h1 element is **transformed and compiled** into React code

JSX Components

- It is possible to embed JavaScript expressions in JSX
- You can put any valid JavaScript expression inside the curly braces in JSX.

```
1
2 import React from 'react';
3 import ReactDOM from 'react-dom';
4
5 const name = 'Sarah';
6 const element = <h1>{name}'s Reading List</h1>;
7
8 console.log(element);
9 ReactDOM.render(element,document.getElementById('root'));
```

JSX Components

- **JSX** expressions become regular JavaScript function calls, which means you can use standard **if** statements and **for loops**.

```
import React from 'react';
import ReactDOM from 'react-dom';

const name = 'Sarah';

function Greeting(user) {
    if (user) {
      return <h1>{name}'s Reading List</h1>;
    }
    return <h1>Just a Reading List</h1>;
}

console.log(Greeting(name));
ReactDOM.render(Greeting(),document.getElementById('root'));
```


29

Exercise 2

30

Exercise 2: Updating React App

1. In your React app, edit the **index.js** file under **src** folder and replace the code with the following:

```
import React from 'react';
import ReactDOM from 'react-dom';

const name = 'Sarah';
const element = <h1>{name}'s Reading List</h1>;

console.log(element);
ReactDOM.render(element,document.getElementById('root'));
```

2. Enter **> npm start** into the command line to see the result

3. Update the variable ‘name’ to your own name and run the project (**> npm start**)

31

Exercise 2: Updating React App

4. Edit the **index.js** file under the **src** folder and replace the code with the following:

```
import React from 'react';
import ReactDOM from 'react-dom';

const name = 'Sarah';
function Greeting(user) {
   if (user) {
     return <h1>{name}'s Reading List</h1>;
   }
   return <h1>Just a Reading List</h1>;
}
console.log(Greeting(name));
ReactDOM.render(Greeting(),document.getElementById('root'));
```

5. Enter **> npm start** into the command line to see the result

32

Class Components

33

Brief Look at Class Component

- This is an example of a BookList component which inherits from the React **component class** or React **component type**

- A component takes in parameters, called **props** (short for properties)

- In a class component the **props** of a class are accessible through the **this** object (line 6)

- In this example the ** ** is **JSX**. It is transformed at build time to **React.createElement('div', … , …)**

```
2   class BookList extends React.Component {
3       render() {
4         return (
5           <div className="booklist">
6             <h1>Books for {this.props.name}</h1>
7             <ul>
8               <li>Half of a Yellow Sun</li>
9               <li>Black Leopard, Red Wolf</li>
10              <li>Born a Crime</li>
11              <li>Americanah</li>
12              <li>Ghana Must Go</li>
13            </ul>
14          </div>
15        );
16      }
17    }
```


34

Brief look at Class Component

- Class components need a **render()** method which will return the React.Elements

- There are currently no plans to remove classes from React. However React creators do not recommend writing class components anymore in favour of **React Hooks**

- It's helpful to recognise the class component syntax and understand how to convert them to functional components

```
class BookList extends React.Component {
    render() {
      return (
        <div className="booklist">
          <h1>Books for {this.props.name}</h1>
          <ul>
            <li>Half of a Yellow Sun</li>
            <li>Black Leopard, Red Wolf</li>
            <li>Born a Crime</li>
            <li>Americanah</li>
            <li>Ghana Must Go</li>
          </ul>
        </div>
      );
    }
  }
```


35

Functional Components

36

Functional Components

BLACK CODHER
CODING PROGRAMME

- Functional components also known as **stateless components** are components that are functions.
- One of the main differences between a **functional** component and **class** components is the syntax.
- functional components do not require you to extend from **React.Component**.
- They are normal JavaScript functions that take a **props** argument.
- They do not have a **render()** method. You return your react element directly as a return object of the method.

37

From Class to Functions

BLACK CODHER
CODING PROGRAMME

- With the introduction of **React Hooks** you can set **state** inside functional components
- Functional components can help with using best practices as it's easier to separate concerns* and you write less code
- Previously, if **state** was needed in a component you would need to create a class component or alternatively **lift state** up to a parent component and pass down via **props**
- It is possible to convert your class components into functional components
- **Functional components** produce less React boilerplate code

38

Checkpoint!

How are you feeling?

RED - I have no idea what you're talking about.

YELLOW - I have some questions but feel like I understand some things.

GREEN - I feel comfortable with everything you've said.

Component Lifecycle

40

Understanding the Lifecycle

BLACK CODHER
CODING PROGRAMME

- Everything follows a lifecycle (e.g. animals, plants). They are born, they grow, and then they die.

- React Components also follows a similar cycle:

  1. Components are created (**mounted** on the DOM).

  2. The components grow (**updated**).

  3. Components then die (**unmounted** on DOM).

- This is referred to as the **Component Lifecycle**.

41

Understanding the Lifecycle

BLACK CODHER
CODING PROGRAMME

- The lifecycle methods are available from **class components** as components defined as classes currently provide more features

- Each of the stages described birth/mount, growth/update and death/unmount have a set of methods that can be overridden in the class

- The only method you must define in a React.Component subclass is called render() all other methods are optional

42

Component Lifecycle

BLACK CODHER
CODING PROGRAMME

Lifecycle Methods (in order of execution)

Mounting (Birth)
constuctor()
static getDerivedStateFromProps()
render()
componentDidMount()

Updating (Growth)
static getDerivedStateFromProps()
shouldComponentUpdate()
render()
getSnapshotBeforeUpdate()
componentDidUpdate()

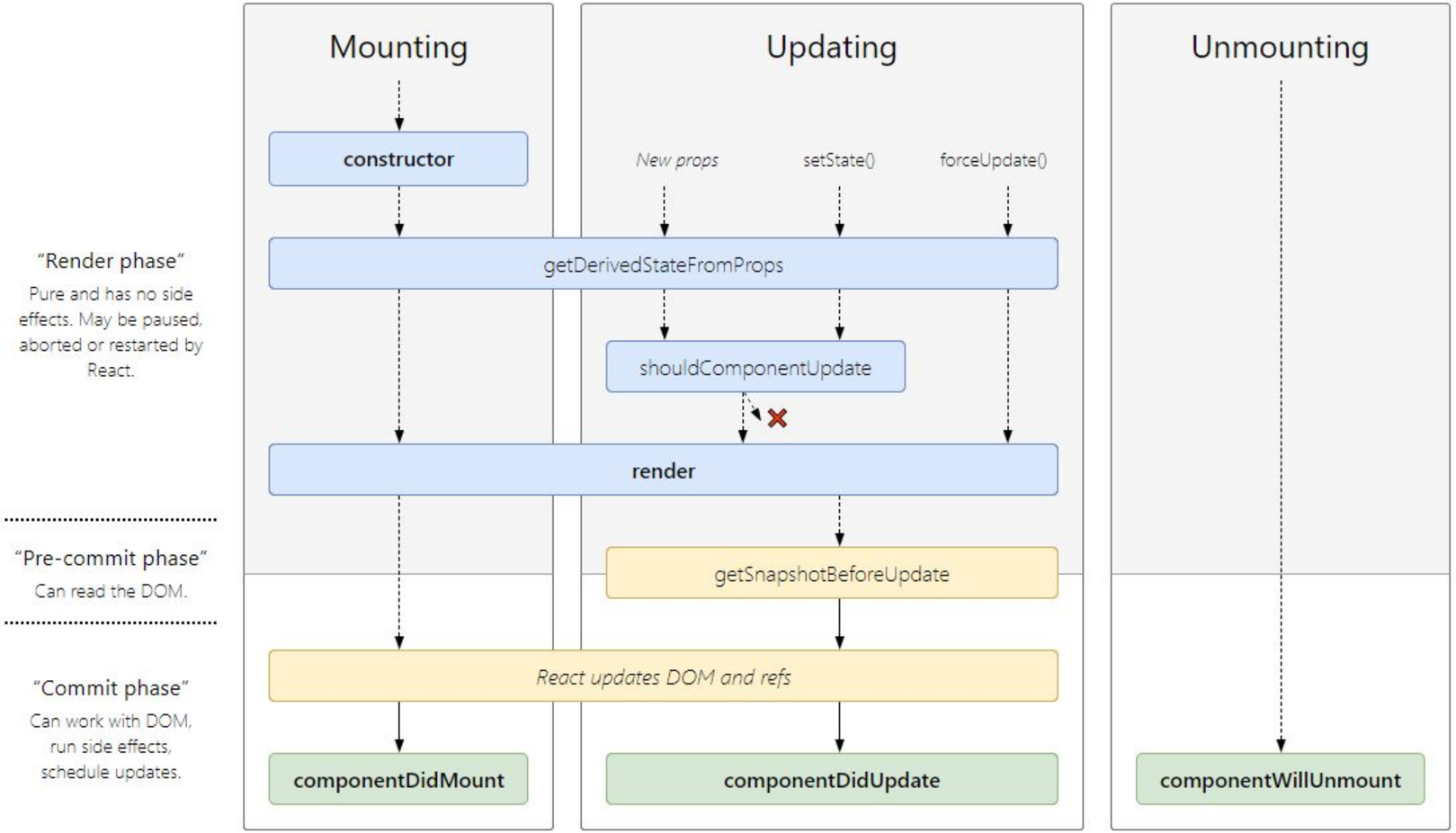
Unmounting (Death)
componentWillMount()

- The life cycle methods can be useful to free up resources taken by the components when a component is destroyed.
- Special methods allow you to call code that can help setup or clear up resources when a component mounts or unmounts.
- The order of execution of the components are important for when they are called.

43

Lifecycle Cheatsheet

https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/



Important Component Methods

- In class components there are a few important methods that are required to implement a component.

- The **`render()`** function is the only method* that is required in a class component. The **`render()`** function should not modify the component state (remain pure).

- The **`constructor()`** is called before a component is mounted. When implementing the constructor for a **`React.Component`** subclass, you should call **`super(props)`** e.g.

```
constructor(props){
  super(props);
  this.state = { started: false};
   this.startGame = this.startGame.bind(this);
}
```


45

Important Component Methods

- The **`constructor()`** is the only place that **`this.state`** should be assigned in a class component

- **`componentDidUpdate()`** is invoked immediately after updating occurs. This method is not called for the initial **`render()`**

46

Properties (Props) Explained

47

Props or Properties Explained

BLACK CODHER
CODING PROGRAMME

- **Props**, which is short for **properties** are arbitrary inputs accepted by functional components as arguments

- The props contains any **attribute** attached to the element when declared. For example, you can declare the Welcome component with the attributes:

```
const element = <Welcome firstname="Faith" lastname="Evans"/>;
```

- The props object will have the following attributes:

```
props {
  firstname : "Faith"
  lastname: "Evans"
}
```


48

Props or Properties Explained

BLACK CODHER
CODING PROGRAMME

- The property can then be accessed in the Welcome function using dot notation e.g. props.firstname

```
function Welcome(props) {
  return <h1>Hello, {props.firstname + " " + props.lastname}</h1>;
}
```

- Props can contain any JavaScript element, function or variable

- Props are read-only, so you **cannot** modify props inside a functional or class component

49

Props or Properties Explained

- All React components must be **pure components or functions in respect of their props**

What does that mean?

- A function should not try and alter it's inputs, it should remain 'pure':

```
//Pure function
function Add(x,y) {
  return x + y;
}
//Bad practice!
function Add(x,y) {
  x = x + 1;
  return x + y;
}
```


50

State Explained

51

State Explained

- React has another special built-in object called **state**, which allows components to create and manage their own data.

- Unlike **props**, components cannot pass data with **state**, but they can create and manage it internally.

52

State Explained

- Example of Using state in a class component:

```
class Welcome extends React.Component() {
 constructor(props){
  super(props);
      //Setting state
      this.state = { firstname: "Donna", lastname: "Summer" }
   }
   render(){
      return <h1>Hello There {this.state.firstname}
{this.state.lastname}!</h1>;
   }
}
```

State Explained

- State is set in the **constructor()** of a component which is called only once when the component is created.

- State should not be modified directly but can be modified with a special method called **setState().**

```
this.setState({
    lastname: "Winter"
});
```

- Changing the state of a React component will trigger a re-rendering of the component (**Not the whole DOM**).

- The **setState**() method triggers the re-render.

54

State Explained

BLACK CODHER
CODING PROGRAMME

- If you use **React Hooks**, **state** can be changed in both **functional** and **class** components.

- Without React Hooks, state can only be used in class components.

- It's important to note that **changing state directly is possible**, but **bad practice** as it will not cause the component to re-render. **So, Don't do this when you want to update state**: `this.state.lastname = "Winter";`

- If you want to set state using props, pass props as the second argument of the **setState()** method e.g.

```
this.setState((state,props) => ({
   lastname: props.lastname + "(for "+ state.lastname + ")"
}));
```


55

Checkpoint!

How are you feeling?

RED - I have no idea what you're talking about.

YELLOW - I have some questions but feel like I understand some things.

GREEN - I feel comfortable with everything you've said.

End of Session 3 Summary

In this session we covered the following:

1. Editing the Create React App
2. Viewing the React elements in a browser window (using Developer Tools)
3. Class components and Functional Components
4. JSX Components Explained
5. Understanding the Component Lifecycle
6. Understanding Props (Properties) and States in React

57